Formal Certification and Compliance for Run-Time Service Environments

Howard Foster, George Spanoudakis and Khaled Mahbub
School of Informatics, City University London
Northampton Square, London
England, United Kingdom
Email: {howard.foster.1,g.e.spanoudakis,k.mahbub}@city.ac.uk

***Abstract*—With the increased awareness of security and safety of services in on-demand distributed service provisioning (such as the recent adoption of Cloud infrastructures), certification and compliance checking of services is becoming a key element for service engineering. Existing certification techniques tend to support mainly design-time checking of service properties and tend not to support the run-time monitoring and progressive certification in the service execution environment. In this paper we discuss an approach which provides both design-time and run-time behavioural compliance checking for a services architecture, through enabling a progressive event-driven model-checking technique. Providing an integrated approach to certification and compliance is a challenge however using analysis and monitoring techniques we present such an approach for on-going compliance checking.**

I. Introduction

The ability to assess the trustworthiness of service-based software systems (SBSS) is increasingly gaining attention [1]. Currently mechanical trust reasoning mechanisms are supported by some basic certification and compliance schemes, with certificates typically measured and awarded against traditional, monolithic software systems. However these mechanisms easily become invalid when a system intends to perform a dynamic run-time adaptation of components. Thus, current certification and compliance mechanisms are insufficient to support the dynamic nature of service-oriented systems. Furthermore, the transition to a cloud-blased deployment of SBSSs requires some higher-level of control for infrastructure and software services, promoting the ability to perform some advanced compliance checking processes for service quality levels and other run-time properties (such as conditional adaptation or discovery).

To aid in this endeavour, our interests focus on providing rigorous yet practical methods for certification and compliance checking based upon the run-time events exhibited by such systems. Services may also be composed statically or dynamically (i.e. at design-time or run-time) and can exhibit behaviour which had not been expected, desired and may lead to violations in service compliance policies. Our approach provides two main areas of compliance checking. First, a design-time compliance that checks aspects of service behaviour design against, for example, agreed orchestration or choreography models. Second, run-time model-checking providing a mechanism to uphold service properties during execution. Our focus is on checking properties in two generic scenarios; (1) in the course of dynamic service discovery with *advertised* service behaviour and (2) in *progressive* service behaviour and *potential* service violations.

Our contribution aims to assist both service engineers and auditors to analyse the compliance levels of services in their run-time environments. For engineers it can provide a mechanism to support adaptation of services based upon these levels, whilst for compliance it enables a measure for certification. We achieve this analysis through a series of algorithms which are implemented to provide a formal rigorous set of software components which may be included in run-time environments or result in visual reports in an analysis workbench. To assist adaptation of design and run-time models we highlight potential violations as a result of comparing run-time event traces and service design specification models. One illustrated example application of this is to enhance the service monitoring with levels indicating when properties should be analysed as bounded measurements.

The paper is structured as follows. In section 2, we discuss a background to services compliance checking and in section 3 we provide scenarios for compliance checking. In section 4, we outline our approach to gather service behaviour details for both statically and dynamically checking service behaviour models built upon a service-monitoring architecture. In section 5, we detail the analysis for our focus on service compliance checking. In section 6, we describe a prototype implementation of the algorithms discussed previously and discuss an initial evaluation. Section 7 discusses related work and finally in section 8 concludes the paper with a summary of present work and future direction.

II. Background

With the increased awareness of security and safety of services in on-demand distributed service provisioning (such as the rapid adoption of Cloud-based resource infrastructures), certification and compliance checking of services is becoming a very important topic of service engineering. Existing techniques tend to support mainly design-time approaches to checking service properties, whilst existing policy-driven run-time monitoring approaches use a policy specification to uphold some compliance rules. This has been realised for example in a Service-Level Agreement (SLA)), however a

formal model-based approach is typically not involved at run-time (e.g. as in PONDER2). Such rigour is needed at various stages in the service engineering lifecycle and execution. This compliance checking may be realised in a Service-Oriented Architecture (SOA) with checkpoints such as the ones described by the CBDI [2] illustrated in Figure 1.

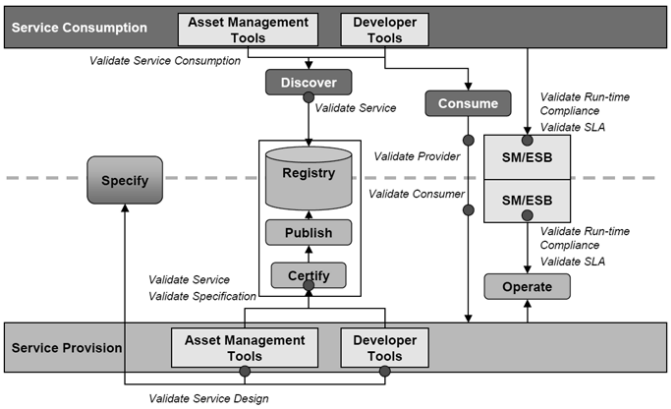

Fig. 1. CBDI Example SOA Policy Compliance Checkpoints

In particular, Figure 1 highlights checkpoints at Service Design, Consumption, Certification and Run-time Compliance (highlighted by little circles in Figure 1). Integrating rigorous static testing and dynamic checking tools constitutes a great challenge in providing greater assurance of services in these areas of service engineering.

A. Service Design and Consumption

A number of verification approaches are aimed towards service design. Our previous work [3] focused on integrating service composition analysis in this area and provided a general approach to design vs. implementation verification. As an overview, the approach offered analysis from several verification property perspectives including design specification against implementation, interaction (co-ordination), obligations (orchestration vs. choreography) and in deployment (resource constraints and process safety). Common to all types of analysis were a series of steps to abstract a combined model of specification and implementation in preparation for analysis. Note that *Service Validation* in this context specifically aims to address that the right service had been built with *Service Verification* assuring that the service is built correctly.

B. Service Certification

Software certification has largely focused on certifying security mechanisms (e.g. ISO/IEC 15408) or product quality metrics (e.g. ISO/IEC 9126*). Our work in certification extends these with assertions for other properties of services and is being carried out as part of the EU funded project ASSERT4SOA [4]. ASSERT4SOA aims to provide a more general service certification framework, notation and architecture for supporting certificates in service discovery and execution. Certification evidence is split into three certification areas: *Evidence-based*, *Model-based* and *Ontology-based* resulting in ASSERT-E, ASSERT-M and ASSERT-O certificates respectively. The three types of certification are used as part of a wider framework to specify and utilise certifications in a service-oriented architecture. In this paper however, we focus on the use of model-based certification for upholding certain service properties, since we consider analysis on formal models. The model that is the basis for verifying that a particular service provides a particular security property (verification model for the rest of this document) will usually be on a lower abstraction level. This implies that there can be several different services with different verification models that have the same ASSERT-M model, as described in [4] and depicted in Figure 2.

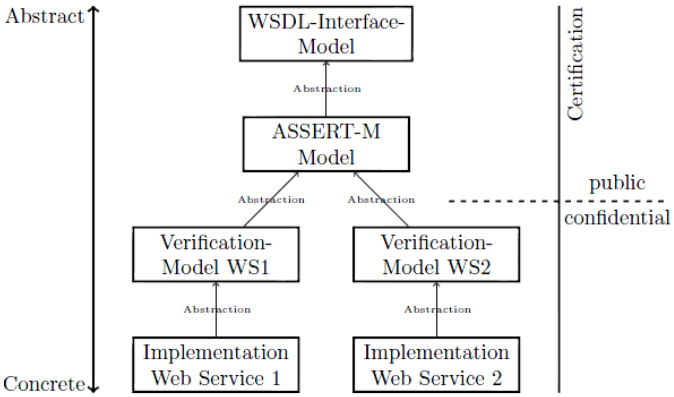

Fig. 2. ASSERT4-M, Verification and Service Models Hierarchy

Depending on the security properties to be specified, the service interface model (e.g. in WSDL notation) may be identical to the behavioural (ASSERT-M) model or may be an abstraction. The implementation of a Web Service can in theory be specified in a formal model as well, however this will usually not be the basis for formal verification. Nevertheless we add it in the figure to demonstrate the complete range of abstraction levels. Note that only the WSDL interface and ASSERT-M models, respectively, can be considered public (being part of a certification reference set).

C. Service Run-time Compliance

Checks for run-time service compliance focus on actual service behaviour and in upholding certain properties throughout the execution environment where the certificate confirms the compliance. One specific example of using this mechanism for monitoring compliance at run-time is based upon the substitution of a service on the basis of some predefined event or property violation. The candidate service exposes its service behaviour at run-time. This can then trigger two forms of analysis. The first type of analysis evaluates the *advertised behaviour* and compliance against a design specification of the service related properties. The second form of analysis considers the actual service behaviour (event-driven) and it's

progressive behaviour against the initial service designs. Both types of analysis rely on monitoring properties of services based upon some specification supplied for compliance measures.

III. Scenarios

The main scenario for our work focuses on comparing run-time service events with agreed design-time specifications of service behaviour and properties with the aim to support the auditing and engineering of appropriate services for service systems. A viewpoint of the auditor maybe to provide assurance that the services implemented do indeed fulfil the properties required in the service environment, whilst an engineer maybe interested in whether the services correctly follow protocol in establishing a service composition.

A. Example

For our scenario we have an example of a booking service described by the state machine in Figure 3. In this example, the state machine specifies that a request for a booking may be declined or accepted based upon successfully processing a payment. If the booking is declined (prior to payment) then the booking is cancelled. Both accepted or declined bookings are eventually replied to confirming the status of the booking request. A series of constraints (cased for example, in an SLA) may state that a request for a booking must always be replied to. Also, by design, there maybe a constraint that only a single payment can be made for each booking. Such properties can become the subject for certification and compliance.

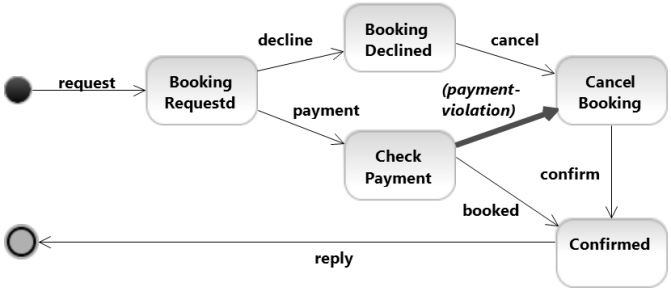

Fig. 3. State Machine highlighting a Property Violation on a Service Trace

This example leads us to ask how model-based compliance and service monitoring could be used in both design-time and run-time execution models to detect and uphold compliance through the execution of services in a services environment.

B. Design-time Compliance

As discussed in section II, there has been considerable research focusing on providing design-time verification techniques for service models. In particular, common properties for verification maybe related to service orchestration, service interactions and overall communication models. If we consider the compliance aspects suggested in Figure 1 then for example, we can validate the service design at this point for behavioural aspects *statically* against agreed service choreography models.

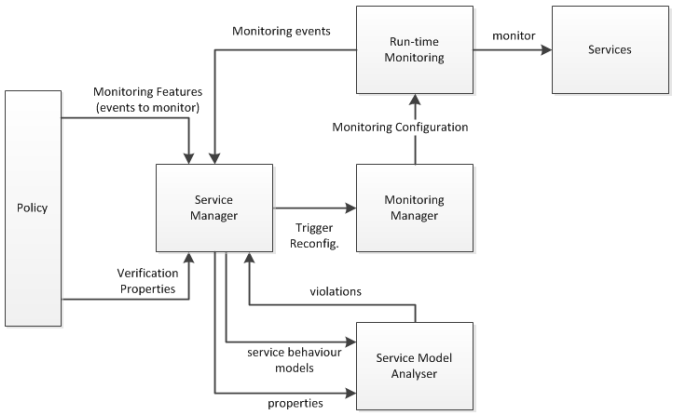

Fig. 4. Architecture for Combining Static and Dynamic Service Analysis

This assists service engineers in validating service designs as part of a collaborative development. Models produced from these steps may also then be *certified* (in the form of an ASSERT-M certificate) and supplied as evidence in service consumption, that certain quality properties are satisfied.

C. Service Run-time Compliance

Further to design-time analysis, there is additional value to combining static service analysis with service monitoring. Such a partnership can aid mechanically checking and detecting levels of service compliance and, through those forms of analysis, to ensure certifications are upheld through the software engineering process and at runtime. Furthermore, run-time compliance checking of service composition activities is crucial to synchronizing appropriate behaviour between processes and upholding properties of service choreography (such as those defined in an SLA policy). The clients of compositions however, may expect different behaviour depending on their individual requests. Therefore the composition must be tested against various scenarios to reflect these different sequences of activities.

IV. Approach and Models

Our approach for undertaking the compliance checking is illustrated in Figure 4 and is described as follows. A policy is supplied consisting of required service properties (behaviour, quality etc) to establish a certain goal. In addition, the policy is supplied along with a set of service monitoring capabilities. The monitoring capabilities are descriptions of service components along with their events and analysis capabilities that may be pulled together to assemble a monitoring infrastructure [5]. Service process descriptions are then supplied (or discovered in service repositories) representing the design-time service composition behaviour (e.g. in WS-BPEL notation). At this stage the policy and service process descriptions are passed to a *Service Manager*. The Service Manager ensures the initial choreography is supported by the service process descriptions by invoking the *Service Analyser* (and by supplying policy, service process models and properties to the analyser). By

default, a property checked is on process deadlock freedom. Once satisfied, the Service Manager triggers the generation of a service configuration by invoking the *Monitoring Manager*. The Monitoring Manager generates a suitable monitoring infrastructure configuration and supplies it to the Monitoring Engine. Events generated from monitoring the services are supplied back to the Service Manager which monitors the events against the rules derived from the service policy and any violations - re-triggering a monitoring configuration if necessary. The reconfiguration of monitoring may enable certain properties which were not critical to the initial service compliance (e.g. unbounded service availability).

A. Models

To prepare for analysis we use some common modelling notations to define the service specifications, design and run-time behaviour.

1) Labelled Transition Systems: We define the behavioural and structural semantics of each service composition artifact in terms of a Labelled Transition System (LTS) [6]. Labels can represent different things depending on the context the system is used in. Typical uses of labels include representing input expected, conditions that must be true to trigger the transition, or actions performed during the transition. We use LTSs to describe the formal behaviour of service specifications, both in design and implementation models. LTSs can be modelled using the Finite State Process (FSP) notation [7] which can be compiled into LTSs using the Labelled Transition System Analyser (LTSA) tool [8]. FSP is designed to be easily machine readable, and thus provides a preferred language to specify abstract processes. FSP is a textual notation (technically a process calculus) for concisely describing and reasoning about concurrent programs. FSP supports a range of operators to define a process model representation.

2) Behavioural Specifications: As we reported in [3] we use the WS-BPEL specification as an example implementation of service orchestrations and leverage the WS-CDL for a specification of service behavioural policy (as a choreography). WS-BPEL defines a series of constructs to describe a service composition process, where a local partner in the composition executes a series of service interactions. Our behavioural mapping of WS-BPEL to FSP groups activities by their related areas in its specification. We use the WS-CDL specification as an example language to describe service choreography and map the constructs of this language to the semantics of LTSs using FSP models. A WS-CDL package consists of choreographies that specify one or more scenarios of interaction activities between different partners. At the choreography package level, general aspects common to all choreographies are defined, for example participant roles, types, channels for communication and information (data) sets. Within each choreography scenario are activities which specify interactions, properties, workunits and finalization steps. Given one or more WS-BPEL processes and a WS-CDL choreography specification, we can model sufficient detail to support the design-time service behaviour analysis. Additionally, a dynamic check of service design can be provided given a similar analysis technique however, the abstract process specifications (in WS-BPEL) can be released on service discovery. However, to perform dynamic run-time analysis we require to capture additional service behaviour detail in the form of service (and other service environment) events.

3) Service Interaction Events: To notify interested clients of service interaction events, our architecture supports a service event structure which is adopted by each service (or distributed event bus mechanism) to report on such events. The event is reported with a number of attributes which are used to model such events and transform them to representations in the behavioural specifications discussed previously. The event structure is illustrated in Figure 5.

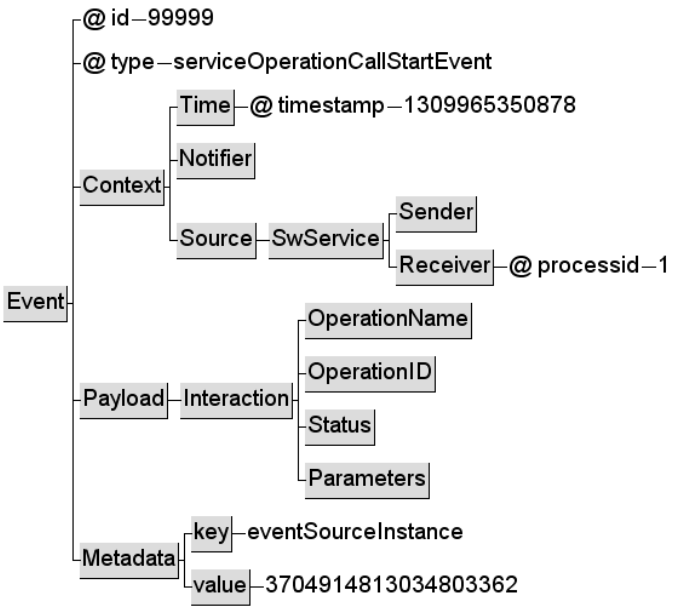

Fig. 5. Structure of a Service Event as XML Elements

As shown in the Figure 5, each event instance contains a unique ID, context, payload and metadata information of the event. Context information comprises a time stamp of the event, a collection time of the event and the source of the event. A payload of the event signifies whether the event is an interaction event or a monitoring result event. In the case of interaction event, the payload specifies the name of the operation that is signified by this event along with the list of arguments of the operation. In the case of monitoring result event, the payload contains the monitoring result for an SLA guarantee term and the reference to the SLA that the guarantee term belongs to. Finally, a list of meta-data, represented as key value pairs, describes additional detail of the event (e.g. the ID of the process instance that produced the event).

4) Monitoring Events and Violations: We use the EVEREST Monitoring Engine [9] to monitor events and violations from rules specified in a service SLA. The properties that can be monitored by EVEREST are expressed in the operational monitoring specification language of EVEREST, called EC-

Assertion [10], which is an XML language based on Event Calculus (a first-order temporal logic language). The basic modelling constructs of Event Calculus (and EC-Assertion) are events and fluents. An event in Event Calculus (EC henceforth) is something that occurs at a specific instance of time, is of instantaneous duration, and may cause some changes in the state of the reality that is being modelled. This state is represented by fluents. To represent the occurrence of an event, EC uses the predicate Happens(e, t, $\Re$(t1,t2)). This predicate represents the occurrence of an event e that occurs at some time point t within the time range $\Re$(t1, t2) and is of instantaneous duration. The boundaries of $\Re$(t1, t2) can be specified by using either time constants or arithmetic expressions over the time variables of other predicates in an EC formula. The EC predicate Initiates(e, f, t) signifies that a fluent f starts to hold after the event e occurs at time t. The EC predicate Terminates(e, f, t) signifies that a fluent f ceases to hold after the event e occurs at time t. An EC formula may also use the predicates Initially(f) and HoldsAt(f, t) to signify that a fluent f holds at the start of the operation of a system and that f holds at time t, respectively. EC-Assertion adopts the basic representation principles of EC and its axiomatic foundation and introduces special terms to represent the types of events and conditions that are needed for runtime monitoring. The EVEREST receives the SLA to be monitored and produces operational monitoring specifications in EC-Assertion by applying the translation mechanisms described in [11].

5) Monitoring Service Properties: In addition to behavioural properties without time restrictions (for example a response always follows a request) unbounded properties (such as availability) cannot be typically monitored at runtime. Since we are interested in monitoring properties that are proposed at design time but must be monitored at runtime we need to add constraints to the property. Unbounded properties, therefore, need to be amended with a time boundary condition. For example the unbounded property **always (send(o1,o2,requestValue) implies (eventually send(o2, o1, sendValue))** which specifies that a request for an operation requestValue is always (eventually) followed by a reply sendValue. Amending as a bounded property the same proposition becomes **always (Happens(send(o1,o2,requestValue), t1 r(t1,t1)) ==> Happens (send(o2, o1, sendValue), t2, R(t1, t1+D)))** where t1 and t2 are time variables indicating the timestamp of the relevant request and response, and D determines the time period within which the property must be satisfied.

V. Analysis

Analysis can be broadly described into two areas; static and dynamic. Static analysis is aimed at the engineer to assist with checking the design time compliance of a service against an agreed policy. Dynamic analysis considers the execution or discovery of services in the execution environment. To address compliance in dynamic analysis we consider advertised behaviour, progressive behaviour and potential violation detection in this area.

Requirement:	Check that partner service implementations fulfil choreography policies.
Input:	One or more service implementations (WS-BPEL) and one choreography specification (WS-CDL). Property as either name of partner role or empty (for all partner roles).
Output:	A set of actions to trace violation or an empty set.
Algorithm:	1. **transform** choreography specification to *chorspec* model 2. **for each** service implementation **do** (a) **transform** service implementations to *impmodels* (b) **map** service interactions to *chorspec* activities (c) **build** *exceptionlist* of implementation exceptions (d) **map** exceptionlist to chorspec *exceptions* 3. **if** *role* **is not empty then** (a) **hide** activities **not equal to** role action in chorspec (b) **set** property **to** role orchestration name 4. **generate** *analysis* model (*impmodels+spec*) 5. **perform** compositional reachability analysis on *analysis* model

Fig. 6. Algorithm for Static Analysis of Obligations

A. Static Compliance Analysis

Static Compliance can be satisfied by taking a service design model and by checking that a set of properties over this model (as set out in a compliance policy) are always satisfied. One such set of properties is an *obligations analysis* to assure that required actions are fulfilled in a service design model. The algorithm for static behavioural compliance is listed in Figure 6. The algorithm takes as input a WS-CDL choreography policy specification and one or more service orchestration implementation in WS-BPEL. The analysis model is constructed by transforming and mapping each implementation (i.e. each partner implementation process within the choreography) and mapping those actions to the choreography model actions. Finally, verification performs a deadlock freedom analysis on the generated analysis model. The output indicates any violations exposed in the service design against a service policy (in the case of that specified in the choreography).

B. Dynamic Compliance Analysis

We cover two types of analysis based upon either the advertised behaviour offered by a service and/or the actual service behaviour based upon a *progressive* accumulation of service interactions over an enactment of a set of choreography obligations.

1) Advertised Behaviour: This is behaviour advertised by a service on discovery. The advertised behaviour can be specified as part of an abstract process (such as the abstract WS-BPEL specification). The abstract process provides sufficient detail to examine the potential role and fit of the service within a wider service choreography. We provide choreography analysis to compare multi-partner service policies and specifically their *obligations* with that of individual partner service implementations. The obligations analysis considers a partner's role in the choreography and checks their obliged interactions set out in the choreography policy. Using the same approach, each partner role in the choreography can be checked. In fact, one of the reported aims of WS-CDL is to

be used as a specification which can be distributed between partners to aid their implementation of service obligations. Hence the analysis can be used to determine whether a service's advertised behaviour is "fit-for-purpose" in the wider choreography being undertaken. An algorithm for advertised behaviour analysis is listed in Figure 7. The output of using the algorithm is as an indicator to whether the service is suitable for inclusion in a service composition.

Requirement:	Check that a substitute service implementation fulfils a choreography policy.
Input:	A behavioural specification (abstract process) of service substitute, a service choreography specification and a property as either name of partner role or empty (for all partner roles).
Output:	A set of actions to trace violation or an empty set.
Algorithm:	1. **set** chorspec to *choreography model* 2. **receive** service substitution *event* 3. **build** *absmodel* from *abstract process* 4. **set** *analysis* model as (*absmodel+chorspec*) 5. **perform** compositional reachability analysis on *analysis* model

Fig. 7. Algorithm for Advertised Service Behaviour Analysis

2) Analysis of Progressive Behaviour: This is the actual behaviour of a service, gathered and accumulated through service interaction events (i.e. service operations requests and responses). A dynamic analysis algorithm based upon service events is illustrated in Figure 8. The dynamic event-based service analysis accumulates a list of service interaction events for each process or service identified by the event. The analysis model consists of a state machine built from the observed run-time state events (rt-model) and the properties to checked. The verification of the progressing behaviour of service orchestration is verified against the service choreography for the service behaviour (as included and registered in a service-level policy).

Requirement:	Indicate properties that may be affected by a violation.
Input:	Progressive model of behaviour (interaction events) and a set of rules for properties (rules)
Output:	A list of properties with impact rating (Green, Amber, Red).
Algorithm:	1) **init** *rtmodel* to *start* 2) **for each** service in choreography **do** (a) **listen** for service *events* model (b) **receive** service interaction *event* (c) **build** service interaction sequence (d) **append** interaction sequence to rt-model 3) **for each** *property* in **rules** (a) **build** *analysis* model (*rtmodel+property*) (b) **perform** compositional reachability on *analysis* model

Fig. 8. Algorithm for Progressive Service Analysis

3) Potential Violations: This provides an impact analysis of the current state of execution. The analysis is enacted when a property is violated and can be used to determine the effect on other properties that have been verified against the initial model. A running example for potential violations takes as input a service design model (or abstract representation of its implementation) and a set of properties derived from a SLA or other type of policy. For example, an SLA may ask that a service request always implies that a state of the system is reached within a certain time period. First, the service design is checked for correctness given a default safety property of deadlock freedom. If no violations are detected then each property is checked against the design model to assure it is upheld. For each path checked in the design model, a list of *property held paths* is maintained.

Requirement:	Indicate properties that may be affected by a violation.
Input:	Progressive model of behaviour (interaction events) and a set of rules for properties (rules)
Output:	A list of properties with impact rating (Green, Amber, Red).
Algorithm:	1) **init** *rtmodel* to *start* 2) **for each** service in choreography **do** (a) **listen** for service *events* model (b) **receive** service interaction *event* (c) **build** service interaction sequence (d) **append** interaction sequence to rt-model 3) **for each** *property* in **rules** (a) **build** *analysis* model (*rtmodel+property*) (b) **perform** compositional reachability on *analysis* model (c) **if** *property* is **violated** (1) **initialise** properties to **green** (2) **set** violation property to **red** (3) **analyse** paths to **violation** (4) **adapt** design model to resolve **violation** (5) **analyse** paths affected by **violation** (6) **identify** properties on **violation path** (a) **set** property status to **amber**

Fig. 9. Algorithm for Assessing Potential Property Violations

The SLA properties and service information are used to construct a monitoring configuration, which is used as input to a service monitor (listening to and observing the events from the configured services). When the monitoring is configured, the service is released and used in one or more composition processes. Events from these services (or processes) are captured and for each process a trace model is constructed and accumulated. Each time there is a service event (that is related to the compliance services of interest) a run-time compliance check is carried out on the properties and any violations detected.

For any violations, the trace to the violation is sent back to the Service Analyser and compared to the paths that uphold the property. If a violation trace is in the acceptable paths then this suggests that the service or composition has not been built correctly, or infact, that the design model is underspecified. However, if the trace is not in the acceptable paths it may be the case that this should be acceptable in the design or in fact, requires an impact analysis. Furthermore, given the knowledge of an accumulated run-time violation trace, it could be the case that further properties are violated given the initial trace that violate properties in the design proof. For example in Figure 10 through an adaptation, a number of payments are now acceptable. However, this then leads to a potential violation in the unbounded property for **booked**.

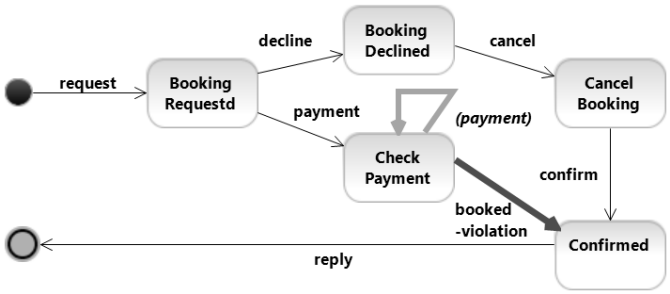

Fig. 10. State Machine Highlighting a Potential Violation after Adaptation

To support monitoring unbounded potential violations (such as booked) we create bounded monitoring rules given an example time period to simulate a constraint and therefore have concrete monitoring rules. Infact, highlighting the properties in a violation, from the run-time trace, leads to the need for resolution. To assist the services engineer we leverage a business process adaptation routine [12] which modifies the design model to support the property violated and then proceeds to highlight other properties of the model which may be affected by this modification. The resolution of these issues is based upon the business problem to be solved by the service solution (i.e. a decision held with the engineer) and is therefore not covered in this paper. However, the mechanism to support such decision making is facilitated by the results of the analysis described previously.

VI. Tool Support and Preliminary Evaluation

We have developed prototype implementations of the service manager, monitoring manager and service model analyser. The components are implemented as illustrated in Figure 11. The prototype accepts two input forms, being one or more events (in the format described in section IV-A and one or more SLA or Policies. The Policies are analysed for rules which are then used to define run-time properties for analysis (for example, a choreography is defined in the SLA in the form of a WS-CDL specification and is defined as the process that must be followed to fulfil a particular role of a specific service). These properties are compiled as state machines (in the form of LTSs) and stored for future reference. Meanwhile, events generated in the run-time environment and placed on a service event bus are analysed for service behaviour. Service behaviour is transformed to the Finite State Process notation and appended to action sequences listed for each service process (identified by a process correlation id in the metadata of an event). As each sequence changes, it is compiled as an LTS and placed in a state machine list (along with the property state machines). When analysis is due (which depends on the context of the property) a complete verification model is built from a composition of process action sequence state machines and the property state machine. Finally, a safety analysis is performed against the verification model to detect any violations of the property specified.

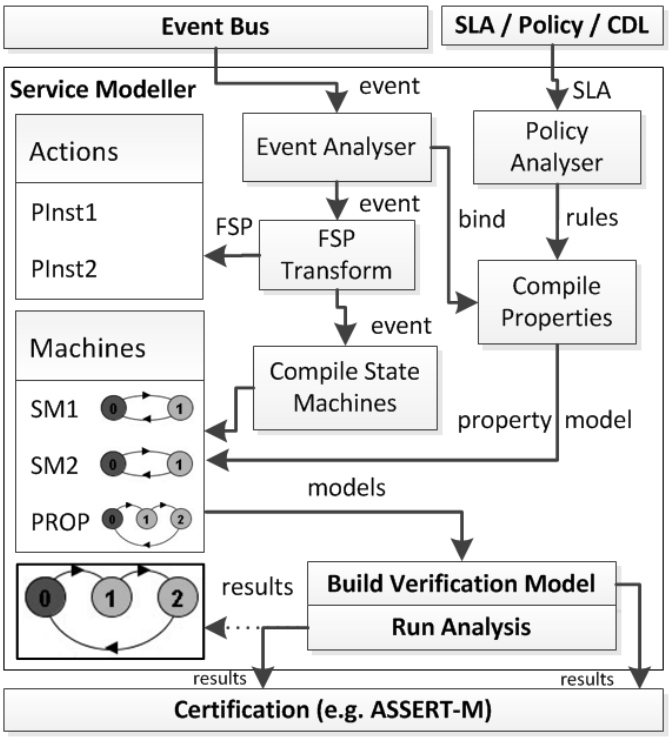

Fig. 11. Implementation of the Event Run-Time Verification Approach

VII. Related Work

Some significant work has been published on modelling and monitoring service events at run-time. In [13] an approach focuses on conformance checking both the fitness and appropriateness of business process execution. For fitness the approach compares a model of the process (design or implementation) and the events of processes recorded in a system log. For appropriateness, metrics are applied to the coverage of activities for one or more processes given a series of events recorded in a log. Thus the conformance measure is an aggregation of measures obtained from fitness and appropriateness analysis. In [14] the authors describe an approach to process mining workflows built from process events. Their aim is to provide a validation mechanism by uncovering and measuring the discrepancies between process design and process executions. Their technique is based upon petri-nets and measuring the distance between nodes in the design and execution models, and also providing an analogue algorithm to reconstruct (split and join) the nodes for corrective path actions.

For monitoring and adapting service compositions, [15] provided an approach to monitoring WS-BPEL processes which, in a similar way to our monitoring infrastructure configurations, defines process monitoring directives for an embedded monitoring manager. Our approach uses known monitoring capabilities of services which exhibit events for service monitoring. We also seek to assist in process adaptation based upon these events. In [16] the authors describe an approach to intercept WS-BPEL service interactions and uses these for monitoring, transforming messages and dynamic

service discovery. The monitoring approach appears to focus only on service interactions, rather than those formed from a broader agreement or policy, measuring invocation specific attributes (e.g. response time, availability). Adaptation is also limited to service discovery and selection (a form of late binding) based upon levels measured in the monitoring steps. For adaptation of processes, whilst upholding consistency rules, [17] describes an approach using graph rewriting techniques to generate suitable change operations. The approach known as ADEPTFlex relies on policies to make suitable changes to workflows in order to achieve consistency in execution. Other work on runtime monitoring focuses on the extension of infrastructure capabilities for capturing and generating the necessary monitoring events (e.g., [18]). Our approach leverages these graph rewriting techniques and operations, however, at this stage in our work we still assume that design governance aids the engineer in order to certify appropriately offered services, which may not necessarily be an mechanical task.

One of the earlier proposals for formal analysis of composition implementations was given in [19]. In this the author suggests that due to the nature of the software assets (the compositions in this case) being deployed to the Internet, that the risk of a bug in such a composition impacts are much greater than that of conventional system deployments. The author of this work has also provided analysis of compositions in terms of those implemented in the Web Service Flow Language (WSFL) [20], which is one of a group of specifications that have been used to create WS-BPEL, and implements a mapping between WSFL and Promela (the language of the SPIN tool) [21]. The work provides a useful reference point on mapping XML schemas (as Web service specifications are typically defined in XML).

VIII. Conclusion

The work described in this paper has focused on a runtime certification and compliance mechanism. To enable this mechanism we provide a framework for analysing service behaviour and policies with properties to uphold, and can be checked for a satisfactory implementation both at design-time and run-time. The novel approach we use is to combine both static and dynamic model-checking techniques with advanced event-monitoring and violation detection. The combination of using LTSA (as a core model-checker) and the EVEREST monitoring engine has so far proved valuable since both are open source and can easily be integrated in an open environment. For future work we plan to formally describe and evaluate process adaptation and optimisation methods to support corrective actions on service property violations. We also wish to explore combining multiple process instances to discover how different implementations of the same service design are affected in the operational environment and to measure the scalability of the approach with regards to performance and distribution.

Acknowledgements

Our work reported in this paper has been supported by the EU project ASSERT4SOA - Trustworthy ICT (ICT-2009.1.4).

References

[1] C. Hang, Y. Wang, and S. M.P., "Operators for propagating trust and their evaluation in social networks," in *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. Budapest, Hungary: ACM, 2009.

[2] L. Wilkes, "Policy driven practices for soa," in *presented at the CBDI SOA Seminar*, 2006.

[3] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "An integrated workbench for model-based engineering of service compositions," *IEEE Transactions on Services Computing*, vol. 3, pp. 131–144, 2010.

[4] ASSERT4SOA, "Advanced security service certificate for soa," in *EU Project ICT-2009.1.4.*, Available from: http://assert4soa.eu/, 2009.

[5] H. Foster and G. Spanoudakis, "Advanced service monitoring configurations with sla decomposition and selection," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, pp. 1582–1589. [Online]. Available: http://doi.acm.org/10.1145/1982185.1982519

[6] R. Milner, *Communication and Concurrency*. NJ, USA.: Prentice-Hall Inc, 1989.

[7] J. Magee, J. Kramer, and D. Giannakopoulou, "Analysing the behaviour of distributed software architectures: a case study," in *5th IEEE Workshop on Future Trends of Distributed Computing Systems*, Tunisia, 1997.

[8] J. Magee and J. Kramer, *Concurrency - State Models and Java Programs - 2nd Edition*. John Wiley, 2006.

[9] G. Spanoudakis and K. Mahbub, "Non intrusive monitoring of service based systems," *International Journal of Cooperative Information Systems*, vol. 15, pp. 325–358, 2006.

[10] G. Spanoudakis, C. Kloukinas, and K. Mahbub, "The serenity runtime monitoring framework," *Security and Dependability for Ambient Intelligence, Advances in Information Security Series*, vol. 15, 2009.

[11] K. Mahbub, G. Spanoudakis, and T. Tsigkritis, "Translation of slas into monitoring specifications," in *Service Level Agreements for Cloud Computing, R. Yahyapour, P. Weider (eds)*. Springer-Verlag, 2011.

[12] Ruopeng Lu, Shazia Sadiq, Guido Governatori, and Xiaoping Yang, "Defining adaptation constraints for business process variants," in *12th International Conference on Business Information Systems*, Poznan, Poland, 2009.

[13] A. Rozinat and W. M. P. van der Aalst, "Conformance checking of processes based on monitoring real behavior," *Inf. Syst.*, vol. 33, pp. 64–95, March 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1316082.1316257

[14] A.J.M.M. Weijters and W.M.P van der Aalst, "Process mining discovering workflow models from event-based data," in *Proceedings of the ECAI Workshop on Knowledge Discovery and Spatial Data*, 2001, pp. 283–290.

[15] L. Baresi and S. Guinea, "Towards dynamic monitoring of ws-bpel processes," in *ICSOC*, ser. Lecture Notes in Computer Science, B. Benatallah, F. Casati, and P. Traverso, Eds., vol. 3826. Springer, 2005, pp. 269–282.

[16] O. Moser, F. Rosenberg, and S. Dustdar, "Non-intrusive monitoring and service adaptation for ws-bpel," in *Proceedings of the 17th international conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 815–824. [Online]. Available: http://doi.acm.org/10.1145/1367497.1367607

[17] M. Reichert and P. Dadam, "Adaptflex: Supporting dynamic changes of workflow without loosing control," *Journal of Intelligent Information Systems*, vol. 10, pp. 93–129, 1998.

[18] W. Ma, V. Tosic, B. Esfandiari, and B. Pagurek, "Extending apache axis for monitoring of web service offerings," in *Proceedings of the IEEE EEE05 international workshop on Business services networks*, 2005.

[19] S. Nakajima, "Model-checking verification for reliable web service," in *Workshop on Object-Oriented Web Services at OOPSLA*, Seattle, Washington, 2002.

[20] F. Leymann, "Web services flow language specification (wsfl 1.0)," IBM, Tech. Rep., 2001.

[21] S. Nakajima, "On verifying web service flows," in *The 2002 International Symposium on Applications and the Internet (SAINT02)*, Nara city, Nara, Japan, 2002.